Reengineering Analysis of Object-Oriented Systems via Duplication Analysis

F. Fioravanti, G. Migliarese, P. Nesi
Dept. of Systems and Informatics University of Florence
V. S. Marta, 3 50139 Florence, Italy
+39 055 4796425, +39 055 4796365, +39 055 4796523
fioravan@dsi.unifi.it, giuseppe@hpcn.dsi.unifi.it, nesi@dsi.unifi.it

ABSTRACT
All software systems, no matter how they are designed, are subject to continuous evolution and maintenance activities to eliminate defects and extend their functionalities. This is particularly true for object-oriented systems where we may develop different software systems using the same internal library or framework. These systems may evolve in quite different directions in order to cover different functionalities. Typically there is the need to analyze their evolution in order to redefine the library or framework boundaries. This is a typical problem of software reengineering analysis. In this paper we describe metrics, based on duplication analysis, that contribute to the process of reengineering analysis of object-oriented systems. These metrics are the basic elements of a reengineering analysis method and tool. A duplication analysis at file, class and method levels have been performed. A structural analysis using metrics that capture similarities in class structure has been also exploited. In order to identify the best approach for the reengineering analysis of object-oriented systems a comparison between the two approaches is described. In this paper a case study based on real cases is presented, in which the results obtained by using a reengineering process with and without the analysis tool is described. The purpose of this study is to discover which method is the most powerful and which time reduction can be obtained by its use.

Keywords: Clones detection, Code Duplication, Object-oriented, Metrics, Reengineering.

1 INTRODUCTION

Software systems are subject to continuous evolution and maintenance activities in order to eliminate programming errors and defects and to extend their functionalities [13].

Frequently, software systems developed from scratch, although starting from the same internal class library or framework, evolve in different directions. During this evolution process, we may need to set checkpoints in order to verify if and when the original class framework or library needs to be corrected or improved in order to simplify the maintenance of applications that have evolved from it. This process often compels to redefine the class library or framework boundaries. This is a typical problem of reengineering analysis.

In general, the planning of a reengineering activity begins with the analysis of the system structure and its functionalities and ends with code modification. Reengineering of a common class framework/library is a typical reengineering activity and often requires modifications to the code under analysis.

During software class framework/library reengineering the ability to localize code duplications can greatly improve the effectiveness of any modifications performed on the code. Since code analysis can be a time consuming activity tools able to improve the speed and effectiveness of this process are desirable [7], [11].

The work presented in this paper focuses on the particular task of reengineering analysis via duplication analysis. Clones represent duplicated codes and therefore can be considered good candidates to be examined in the reengineering process of a class library or framework and its beta applications. Clone Detection in medium/large software systems has been investigated in the past, as in [2], [3], [5], [8],[10]; while clone elimination has been presented in [1], [8]. Several techniques have been reported in the literature to approach the clone identification problems [1], [3], [4], [8], [12], and several results have also been presented [1], [2], [3], [4], [5], [6], [9]. In particular, several approaches give only information on the amount and location of clones [3], [5], [11], [14], while other works present results also on the aspects of clone similarities [1], [2], [10].

As other studies have shown [2], [10], software systems having a large size in terms of lines of code, cannot be manually analyzed without considering specific metrics. An automatic mechanism to determine where the more relevant duplications are present, is a recommended way to proceed. In this paper, the duplication analysis is performed by the tool TREND (Tool for ReEngineering aNd code Duplication) that allows detecting clones or parts of code that are very similar. The tool has been designed paying particular attention to object-oriented systems realized especially in C++. It allows detecting

0-7695-1050-7/01 $10.00 © 2001 IEEE

duplication at the level of files, classes and methods and producing several high-level metrics for duplication analysis, increasing the detection's power with respect to approaches already present in the literature. The presence of high level metrics increases the tool power and simplifies the process of analysis of the huge amount of data generated by an automatic tool for duplication detection and thus of reengineering analysis.

In this paper, the approach followed is to adopt a specific set of metrics for duplication analysis. These metrics are the basic elements of the reengineering analysis method and tool. A structural analysis has been also performed by means of metrics that capture similarities in the class structure.

The research performed, besides the implementation of the tool TREND, has involved the definition and validation of: (i) structural and functional metrics for detecting code duplication at file, class and method level; (ii) algorithms for code preprocessing and for code duplication detection to be adopted in the code analysis phase to produce metric values.

Several different approaches to code duplication have been considered (duplication detection at file, class, method; structural similarities among classes, etc.). In this paper, the comparison among these approaches has been reported in order to show the best approach for reengineering analysis of object-oriented systems. The approach suggested has been validated against real cases. In particular, this paper reports a case study in which the results obtained by performing the reengineering process with and without the TREND tool are discussed and compared. A skilled object-oriented team has reengineered a software suite of programs, without the aid of the TREND tool, while at the same time an automatic analysis to detect code duplication by means of the TREND tool has been carried out. The comparison between the two approaches has been performed in order to highlight the method's power and the time reduction that could be obtained with the tool adoption. An *a-posteriori* analysis of the reengineering performed by the team has also been reported in order to verify if all the duplications were detected by manual code inspection.

This paper is organized as follows: in Section 2 the general problem description, the preprocessing and code analysis algorithms adopted are reported; Section 3 reports the adopted metrics for duplication analysis; Section 4 reports the reengineering analysis manually performed and comments on this process; in Section 5 the analysis performed by the tool TREND on a case study is reported together with obtained results and comments; Section 6 reports an a-posteriori analysis of the manually performed reengineering, while in Section 7 conclusions are drawn.

2 A SPECIFIC REENGINEERING ANALYSIS

Problem Description

In this paper, a problem of software reengineering is analyzed and discussed together with techniques for its management. The specific problem is related to the reengineering activity for maintaining or building class framework or class libraries.

Typically class frameworks are generated from a common framework of several applications developed in the same company. Under these conditions, it is frequent to take decisions for the inclusion or not of other components in the common framework or library. This process is typically manually performed. A similar problem occurs when two or more software systems, originated in the same application domain or from the same library or framework, have evolved in different directions during their life.

In order to increase the maintainability of these systems, it can be often necessary to extract the common part from the systems in order to create or reengineer the class library or framework to be shared among them and other applications of the same software company.

This process is summarized in Fig. 1, in which the systems A and B have been evolved from a common application domain or class library or framework. For this reason, they share a common part of the class framework which has been partially modified during the evolution. The reengineering process needs to merge the common part in a single class framework that will be shared between the new versions of the systems being created.

This described reengineering process covers several different aspects such as:

- structural analysis of the classes of the evolved system in order to identify similar classes in terms of data structure definition (i.e., attributes scope and types). Classes may have different names but modeling the same entity of the application domain. Classes may have attributes with different names and identical types that model the same class features;

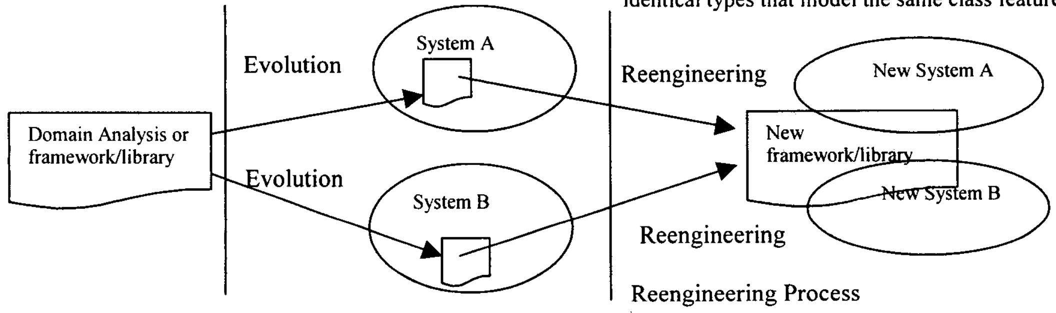


Fig. 1. Reengineering process for software system evolving from the same analysis domain or framework library.

- detection of functional duplications among classes of the systems in order to detect both self-duplications and duplications between the two systems;
- redefinition of the boundaries of the class library or framework, that is the creation of a new library on the basis of the shared classes or functionality of the evolved systems or the improvement of the present library;
- Generalization of the library classes in order to make them more reusable by other applications sharing the same application domain.

In order to suitably manage the reengineering process, several activities have to be performed on code after a very deep analysis of the system structure and functionality. At low-level the needed operations can be (i) deletion of methods or classes, (ii) modification of method or classes, (iii) moving of method or classes in the library or framework. All these activities deal with code manipulation, and therefore a program understanding and a careful changing plan have to be performed. During this reengineering process, a large amount of time is spent in looking for identical or quite similar classes and methods. After this work a phase of reengineering analysis is needed. In that phase, the detected duplications and similarities are organized on the basis of their relevance in the system according to prefixed goals. This means that a very complex and long work of analysis of the data produced from the duplication analysis process is needed. Finally, the decisions about the reengineering actions to be performed are taken on the basis of the (i) results of the duplication analysis, (ii) structural analysis at class level, and on the (iii) planned goals to obtain the final result.

In order to reduce the time for the reengineering analysis, tools for detecting duplications can be used. Most of them produce a very huge amount of information and this cannot be easily manipulated to identify the real system problems. For this reason we have defined a set of metrics and a methodology for shortening the process of duplication and data analysis of the reengineering analysis. The proposed process covers the phases of duplication detection with metric estimation, and the corresponding analysis of produced results. It can be used for fast identification of classes, files and methods that have to be manipulated in order to reach the final results. The process also allows eliminating or reducing at an acceptable level the duplication among the classes involved in the systems under analysis.

Preprocessing and Duplication estimation

Even if the duplication analysis can be performed by using one of the several algorithms, which are present in the literature, new algorithms have been produced to speed up the processing time. The real problems are (i) the code preprocessing, (ii) the extraction of high-level features and duplication related metrics that are on the basis of the reengineering analysis, and the (iii) identification of the most critical problems. In all of these, the solution proposed can be of great help for shortening time and automating the process.

In order to make more effective the code duplication analysis, a preprocessing phase is recommended. This phase tries to eliminate all the noise factors (comment, blank lines, different formatting styles, etc.) which are typically present in the source code. Since the code has to be analyzed line by line, it is also possible to extract structural information, such as the number and the types of class attributes, the number and structure of methods, etc.

The tool TREND uses, for the preprocessing, the algorithms presented in Tab.1, which are typically executed in the reported order.

NAME	*DESCRIPTION*
SPA (Source Processing Algorithm)	Eliminates comments and preprocesses instructions by the means of GNU C++ preprocessor.
SFA (Source Formatting Algorithm)	Writes one instruction per line considering each brace as a single instruction.
VSA (Variable Substitution Algorithm)	Substitutes the variable names with the corresponding type. This operation make it possible to detect as clones fragments obtained by cut, paste and renaming operations. See also [5].

Tab. 1. Preprocessing algorithms with descriptions.

The detection of duplicated fragments highlights groups of consecutive lines of each file/class/method that are present also in other files. The adopted algorithm is a general-purpose algorithm for duplication detection applicable not only to C++. The algorithm has been enforced into the tool and allows the extraction of metrics described in the next sections.

After the preprocessing phase, the code is independent of the formatting and of variable names. This allows performing a more precise code duplication detection and analysis. This phase prepares the code to be correctly processed in order to extract code duplication information. The duplication algorithm is capable to automatically perform the following activities:

- Metrics evaluation with consumptive value at system, file, class and method levels;
- Identification of the sequences (fragments composed by consecutive lines) of duplicated lines;
- Identification of structural similarities at class level;
- Production of metrics and flow chart of methods and functions.

3 METRICS FOR DUPLICATION ANALYSIS

In order to quantify the similarities among classes at code and structural levels several specific metrics have been defined. Let us now suppose to have a software system composed of a set F of files, these contain in turn a set C of classes and a set M of methods. Thus, we have $(X_i, X_j) \in \{F \times F \cup C \times C \cup M \times M\}$, where a couple (X_i, X_j) is an element considered in the duplication analysis.

On this basis, the following metrics have been defined:

- NL_i: Number of Lines of entity X_i,

- $NLID_{ij}$: (Number of LInes Duplicated) number of code lines of X_i which are also present in X_j
- $NLIDS_{ij}$: (Number of Lines In Duplicated Sequences) length of the sequence of lines of X_i that are also present in X_j, A sequence of lines is a set of three or more consecutive lines.
- IID_{ij} : (Identity Index of Duplication) equals to $100\ NLD_{ij}/NL_i$: Duplication index of X_i with respect to X_j.
- IID_S_{ij} : (Identity Index of Duplication in Sequences) equals to $100\ NLIDS_{ij}/NL_i$: Duplication index of X_i with respect to X_j, by considering only the sequences of duplicated lines.

Similar metrics have been defined also for the symmetric couple (X_j, X_i).

An index of the duplication at system level can be easily calculated by using SI (System Identity) metric:

$\forall i,j \in S_1 \cup S_2$: $SI = mean\{IID_{ij}\}$

In order to have a consumptive metric at file, class and method levels, the Harmonic Mean (HM) among IID_{12}, IID_{21}, IID_S_{12} and IID_S_{21} metrics has been adopted. The HM of a set of non null values $A=\{a_1, \ldots, a_n\}$ is defined as:

$$HM = \frac{1}{mean\left\{\frac{1}{a_1}, \ldots, \frac{1}{a_n}\right\}}$$

HM definition gives greater weight to lower terms. Note that HM is always lower than the arithmetic mean. This is an useful feature to produce more conservative estimations. In our estimations HM is also a percentage since it is the mean of values that are percentages.

Some other metrics have been defined at class level for the analysis of structural similarities among classes:

- NAL_i: Number of local attribute of the i-th class;
- $NALST$: (Number of Attribute Locally defined with the Same Type) number of identical in type attributes between two classes, independently of the access qualifier (i.e., private, protected and public);

Several other metrics at system level have been adopted to take in account the different measurable aspects of an entire system.

- *TNAL*: Total Number of Attributes Locally defined;
- *TLOC:* Total number of Lines Of Code;
- *NCL:* Number of classes;
- *TNML:* Total Number of Method Locally defined.
- *Nbyte:* Total Number of bytes of the source files of the system;
- *Nfile:* Total Number of source Files of the system.

4 A REENGINEERING PROCESS

In this section, the case study of a real system is presented. The system under analysis has been developed at the Department of Systems and Informatics of the University of Florence with CESVIT (Center for Software Quality) for project TAC. It has been completely written in C++ adopting the object-oriented paradigm. The main functionality of the system deals with software quality evaluation of object-oriented systems. Its main purpose is the visualization of graphical diagrams (such as histogram and Kiviat) and tables summarizing the metrics values calculated by other tools. The tool is named MWB/VM (that stays for Metric WorkBench and View Manager). The system is well suited for applying the above described reengineering process, since:

- The first version of the system was comprised of 2 applications (MWB and VM). The two programs have been developed by different teams but operate in the same context and share the same applicative domain analysis. Several classes covering the same functionality are present in both systems. Note that the two teams started with a kernel or framework with functionally similar classes and the analysis and design were performed by the same person for both the applications.
- The two systems needed a reengineering to collect in a unique class library or framework common classes and functionality.

The reengineering of these applications for reorganizing the common class framework, was manually performed by skilled people with the support of a simple tool for duplications detection, without preprocessing phase and without metrics estimation with 1 man month spent in analyzing the classes in order to perform reengineering activities.

Results of the reengineering process

In order to distinguish the system before and after the reengineering, the following names have been assumed: MWB/VM *old* for the version before the reengineering (a prefix MO_ has been adopted for all the involved files) and MWB/VM *new* for the version after the reengineering (a prefix MN_ has been adopted).

The target of the reengineering was to obtain two executable files sharing a common class framework reducing or possibly eliminating the redundant and duplicated code. In the following, the reengineering process adopted for passing from MWB/VM old to the new version is described. The team that has operated the reengineering has identified the functionality offered by each class and by each method of the several classes. This has been performed by analyzing the source code and the available documentation in order to highlight the common functions and the duplication level. This operation has been manually carried out with the aid of a simple tool for code duplication and spending a quite large effort (6 man/months).

In Fig.2, the classes of the system under analysis are reported in alphabetic order.

MO_VM	MO_MWB
AddMetricDialog	_CustomMetricList
Attribute	_CustomMetric
Class	_MetricMember
Class_Custom_Metric_Parser	Attributo
ClassDialog	Class
Container	Contenitore
Custom_Metric	Contenitore_Value
Dato	DrawList
Dato_Container	ErrorDialog
ErrorDialog	File
File	Global
Function	GnuDialog
Global_Variable	GnuPlotDialog
InfoOpen	Info
Line	InfoDialog
Messaggio	InfoOpen
Method	Line
Method_Called	LISTA
Method_Called_By_Method	Lista_TAC
Metric	Lista_UKDM
Metric_Container	ListMetric
Metric_Value	Method
Metric_Value_Container	Metriche
MetricInfoDialog	MyTopAppWindow
New_Class_Metric	NLOCDialog
New_Sys_Metric	Parametro
Parent	PlotDialog
PlotDialog	ProjectInfoDialog
Procedure_Called	Single_Metric
ProgressDialog	Sistema
Selected_Metric	UkdmDialog
System	Value
System_Custom_Metric_Parser	Variable
Value_Container	VMDialog
VarDialog	VMGnuPlotDialog
Variable	
View	
View_Container	
ViewDialog	
ViewTopApp	

Fig. 2. Classes of MWB/VM system before the reengineering

A preliminary analysis of Fig.2 highlights that some classes of MO_MWB have the same name of those belonging to MO_VM; i.e., *File* and *Line* of MO_list.hxx are present also in MO_VM.hxx, such as *Class, InfoOpen, ErrorDialog, Method, PlotDialog, Variable* are in MO_MWB.hxx and MO_VM.hxx. Several other classes have very similar names (i.e., an Italian Translation of the corresponding English term, etc.) such as *Attributo, Contenitore, Contenitore_Value, Metodo, Metriche and Sistema* of MO_MWB.hxx, with respect to *Attribute, Container, Value_Container, Method, Metric, System of* MO_VM.hxx. This kind of analysis cannot guarantee that the classes are equal or partially duplicated, but it has been a good starting point for the analysis performed by comparing the classes with an editor in order to verify if the name matching corresponds also to the source code partial matching.

In Tab.2, a comparison between the two systems has been reported. It can be noted that the total number of classes for the new version is decreased, highlighting that a certain level of duplication were effectively present in the system. Note that the reengineering process, in order to produce a reusable library of classes, has produced a different organization of the source code augmenting the number of files from 7 to 10. The increment in the number of bytes is due to the addition of several comments that are not considered in LOC (that is decreased). Note that several classes were collected together in the same file, while a reengineering process based on the building of a reusable library should consider to place one class per file in order to better maintain the library.

System	*NFile*	*NCL*	*TNML*	*TNAL*	*TLOC*	*NBYTE*
MWB/VM old	7	76	589	674	12.016	389.921
MWB/VM new	10	62	530	640	11.708	415.659

Tab. 2. MWB/VM systems status before and after the reenginering process.

In Tab. 3, a detailed analysis of the operations performed on the system classes after the reengineering process is depicted. In particular, comparing these values with NCL, TNML and TNAL reported in Tab. 2, it could be deduced that of 14 classes with a mean of about 4-5 methods with 5 lines of code and 2-3 attributes have been removed. This sentence is not completely correct since 20 classes have been modified, and some lines of code have been added in order to rearrange the functionality.

OPERATION PERFORMED	*NUMBER OF CLASSES*
Deleted	14
Modified and moved in the library	13
Modified	7
Moved in the library	10
Unchanged	32

Tab. 3. Details of the operation performed on the system.

Fig. 3, depicts the modifications performed by the reengineering team. In a more detailed manner, these operations can be summarized, at file level, as follows:

- A library identified with MN_sys_dati.cxx has been produced;
- MO_list.hxx has been deleted and the corresponding parts in MO_VM.cxx/hxx have been moved in the library;
- The part of MO_MWB.cxx that was duplicated in MO_VM.cxx has been eliminated and, after some small modifications, the part of MO_VM.cxx has been moved into the library;
- Some other similar changes have been performed on other files, with a smaller impact with respect to the system size;
- The library has been used by both the new versions of VM and MWB.

Criticism of the Manual Reengineering Process

An a-posteriori analysis of the performed reengineering has highlighted that not all the possible duplications have been eliminated. This fact has been shown by applying the tool TREND for duplication detection on the new versions of MWB and VM. The reengineering team has failed to perform some activities that can be summarized as follows:

- Classes InfoOpen and ErrorDialog are still present in the two executables and especially for InfoOpen (that is similar at 95% between the two executables) a deletion of one copy of the class and the modification and moving of the other in the shared library should be performed;
- The class hierarchy has not been modified in depth and therefore inside the library some classes such as New_Sys_Metric and New_Class_Metric, are quite similar. The same situation can be highlighted also for VMDialog and UkdmDialog classes;
- Classes as System, Class, Method, Function and Variable have still a lot of similar methods and the use of a template can eliminate a lot of duplications;

FILE1	*FILE2*	*IID_S$_{12}$*	*IID_S$_{21}$*	*HM*
MN_VM.cxx	MN_sys_dati.cxx	24	41	37
MN_MWB.cxx	MN_VM.cxx	34	23	36
MN_MWB.hxx	MN_VM.hxx	35	24	33
MN_MetricDialog.cxx	MN_Vdialog.cxx	37	16	29
MN_MWB.cxx	MN_sys_dati.cxx	19	18	24
MN_MWB.cxx	MN_Vdialog.cxx	15	23	24
MN_MWB.hxx	MN_sys_dati.hxx	17	23	23
MN_MWB.cxx	MN_MetricDialog.cxx	9	49	23
MN_MetricDialog.cxx	MN_VM.cxx	29	9	22
MN_MWB.cxx	MN_gpro.cxx	13	22	22
MN_MetricDialog.cxx	MN_gpro.cxx	32	10	21
MN_Vdialog.cxx	MN_VM.cxx	24	9	20
MN_gpro.cxx	MN_sys_dati.cxx	20	13	20

Tab.4. Residual duplication after manual reengineering. Files with HM ≥ 20% are reported.

The duplication analysis at file level has highlighted that a certain level of duplication (also if HM is lower than 40%) is still present in the system. In Tab. 4, the duplication level in the new system is reported, confirming that a certain level of duplication is still present in the system. For brevity, only the files with HM greater or equal to 20% have been reported in the table. It can be noted that a part of VM is also present in the library, and MWB and VM share some functional parts (these lines have been marked in gray on the table).

The results of this analysis has also been presented to the team that performed the manual reengineering, and the results obtained confirmed the impression they had at the end of the reengineering that probably not all possible actions were taken during the manual reengineering, since a so deep analysis is quite impossible to be performed by hand.

5 TREND BASED REENGINEERING ANALYSIS PROCESS

In this section, the reengineering analysis and process performed by using the tool TREND is reported. The analysis can be performed at file and class levels. At the class level the tool allows one to take into account the structural similarities and the duplications among methods. As a result of the reengineering analysis some interesting suggestions on what should be performed on the basis of the tool TREND for reengineering the applications are reported. Note that about 8 person- days have been spent to process and analyze the results of the tool, since the row data needed an organization in spreadsheets in order to provide graphs and all the other stuffs necessary for the evaluation of the system duplication. The analisys of the results needed about 2 person-days

Preprocessing

Such as previously reported, the preprocessing phase has a strong role in the duplication identification. Several experiments have been carried out for analyzing the effects of the different algorithms of preprocessing. In Tab.5, a summary of the different strategies is reported.

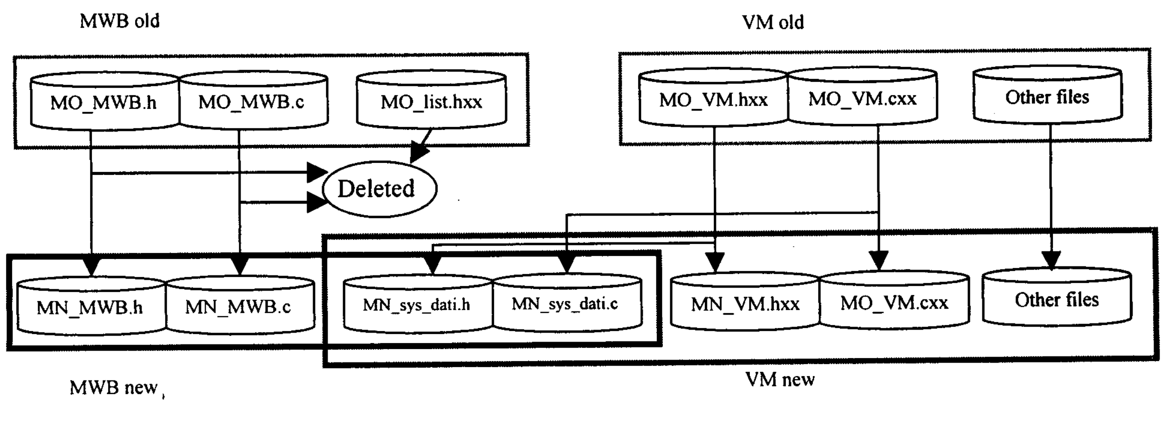


Fig. 3. Reengineering process performed by the team

Note that different values of the duplication index, SI, have been obtained for different preprocessing algorithms. The highest value of duplication has been obtained by applying all steps reported in Tab.1. The values produced by the algorithms have been compared by those obtained by a manual inspection. The comparison has confirmed the results obtained by the application of SPA, SFA and VSA algorithms.

Algorithm	*SI*
No Preprocessing	6.88
SPA	8.38
SFA	10.63
SPA+SFA	13.50
SPA+SFA+VSA	17.25

Tab. 5. Duplications at system level by preprocessing strategy, where SI the System duplication Index.

From the analysis of Tab.5, it is evident that a great improvement to the duplication detection is obtained by substituting variable names with the corresponding types (VSA algorithm). The application of this algorithm has increased the recognition capability of the tool from 13.5% to 17.25% at system level. This fact means that this step is quite relevant to obtain a good duplication analysis, since the name replacing during cut and paste duplication is a quite frequent operation that otherwise would masquerade the real duplication level. For this reason, the old version of the system has been analyzed by using code preprocessed by SPA, SFA and VSA algorithms.

File Level Analysis

The first analysis was focused in the identification of duplications between the old versions of MWB and VM at file level

MWB File (1)	*VM File (2)*	*IID_{12}*	*$IID_{12}UN$*	*HM*
MO_list.hxx	MO_VM.cxx	85	83	70
MO_list.hxx	MO_Vdialog.cxx	*60*	35	35
MO_MWB.cxx	MO_VM.cxx	*56*	31	33
MO_MWB.hxx	MO_VM.hxx	*54*	34	33
MO_list.hxx	MO_gpro.cxx	*57*	28	30
MO_list.hxx	MO_VM.hxx	*52*	26	25
MO_MWB.cxx	MO_VDialog.cxx	43	12	15

Tab. 6. Comparison at file level between MO_MWB and MO_VM systems. Couples having HM > 14%.

In the first two columns of Tab.6, the files related to MWB and VM system before the reengineering are reported, while in the other columns the functional duplication metrics values obtained at file level are reported; Note that only couples of files with HM greater than 14% have been reported. By the analysis of Tab.6, it can be highlighted that MO_list.hxx is duplicated at 85% (IID) and at 70% (considering HM) in MO_VM.cxx. This fact suggests that a strong duplication exists between these files. Note that the comparison has also been performed between a header file, list.hxx, and several source code files, because the header file contained all the methods as inline. In particular, MO_list.hxx is partially duplicated in MO_VM.cxx, MO_Vdialog.cxx, MO_gpro.cxx, and MO_VM.hxx files, while MO_MWB.hxx and MO_MWB.cxx has a partial duplication in the corresponding MO_VM files. This fact confirms the analysis performed by the expert team that has totally deleted MO_list.hxx and partially deleted MO_MWB files moving the common parts of the corresponding MO_VM files into the class library.

In order to have a clear picture of the comparison among all files, the histogram of HM metric for all the possible couple of files has been depicted in Fig.4. Note that in the figure the duplication of files belonging to the same system is also considered, while in the previous table only the comparison among files of different systems is taken into account. A threshold of 20% has been identified for extracting possible duplications at file level. This conservative decision reduced the number of modules to be analyzed at class or method level. By checking the previous table it can be noted that only 6 couples of files reached this value.

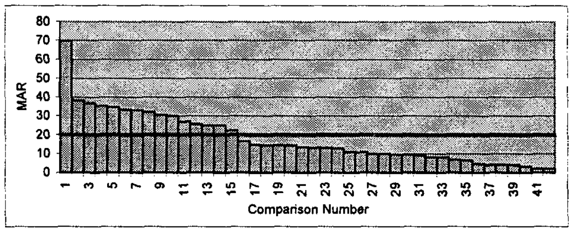


Fig. 4. Histogram of HM values for comparison at file level.

Class Level Analysis

The identification at file level is too coarse to verify whose classes present a strong duplication, then an analysis at class level has been also performed.

The TREND tool can automatically determine the duplications at class level, such as previously performed at file level. An analysis at class level can give more valuable suggestions on how the duplications are distributed.

The analysis at class level can be performed by taking into account two different aspects of class similarities: structural similarities (that is the attributes number and types) and functional duplication (that is how the code is duplicated in methods). These two aspects can give complementary suggestions on how the reengineering has to be performed.

By analyzing the structural similarities (as depicted in Tab.7), it has been noted that several classes have the same structure in terms of attributes. Most of them have a very low number of attributes, in this case it is not correct to automatically states that they are similar or identical. On the other hand, some other classes have a very similar structure, even if not identical, with a larger number of attributes that confirm the detection of structural duplications.

It can be highlighted that a large part of possible duplications is not between the MWB old and VM old, but they have a self-duplication as highlighted by the first

9 rows of Tab. 7.

$CLASS_1$ (FILE)	*$CLASS_2$ (FILE)*	*NAL_1*	*NAL_2*	*NALST*
New_Sys_Metric (MO_VM.hxx)	New_Class_Metric (MO_VM.hxx)	74	75	74
Function (MO_VM.hxx)	Class (MO_VM.hxx)	13	12	11
System_Custom_Metric_Parser (MO_VM.hxx)	Class_Custom_Metric_Parser (MO_VM.hxx)	23	31	23
Function (MO_VM.hxx)	Method (MO_VM.hxx)	13	18	13
Method (MO_VM.hxx)	Class (MO_VM.hxx)	18	12	12
Contenitore (MO_MWB.hxx)	Contenitore_Value (MO_MWB.hxx)	2	3	2
Contenitore_Value (MO_MWB.hxx)	Global (MO_MWB.hxx)	3	2	2
Contenitore_Value (MO_MWB.hxx)	Metriche (MO_MWB.hxx)	3	2	2
Contenitore_Value (MO_MWB.hxx)	VMDialog (MO_MWB.hxx)	3	2	2
PlotDialog (MO_MWB.hxx)	Variable (MO_VM.hxx)	5	4	4
Attributo (MO_MWB.hxx)	Variable (MO_VM.hxx)	3	4	3
Method (MO_MWB.hxx)	View (MO_VM.hxx)	6	6	5
Attributo (MO_MWB.hxx)	Parent (MO_VM.hxx)	3	2	2
Contenitore_Value (MO_MWB.hxx)	Global_Variable (MO_VM.hxx)	3	2	2
Info (MO_MWB.hxx)	Parent (MO_VM.hxx)	3	2	2
InfoDialog (MO_MWB.hxx)	Variable (MO_VM.hxx)	6	4	4
LISTA (MO_MWB.hxx)	Container (MO_VM.hxx)	2	3	2
Method (MO_MWB.hxx)	Variable (MO_VM.hxx)	6	4	4

Tab 7. Structural comparison among classes (HM >= 80). Self-duplications inside the same system are highlighted by gray rows.

The analysis at class level can produce interesting information for the reengineering phase, not only at structural level, but also at functional level. In Tab.8, the functional duplications between MWB and VM classes are reported, and in particular in the first two columns are reported the classes of MWB and VM before the reengineering, while the other columns contain the values of the duplication metrics. Only classes with IID_{12} values greater than 60% are reported. By analyzing the metric values, classes that need to be reengineered can be easily detected. Some classes of the two systems have a percentage of duplication greater than 90%, such as File, Line and InfoOpen class. This fact suggests that a large part of methods have the same source code and then the same functionalities. The analysis at method level, not reported for brevity,. has confirmed this first impression. The two systems have several classes that are very similar and also some functionalities have been replicated by cut and paste technique in other classes. The preprocessing phase has allowed to detect that some methods have been duplicated with internal variable renaming.

The self-duplication of VM at functional level is highlighted also in Tab.9, in which the duplication of several of the previously analyzed classes performed among methods (functional) is reported. The same analysis has been conducted also for MWB producing the results reported in Tab.10.

MO_MWB (1)	*MO_VM (2)*	*IID_{12}*	*IID_{21}*	*IID_S_{12}*	*IID_S_{21}*
File	File	99	95	99	92
Line	Line	96	96	96	96
InfoOpen	InfoOpen	91	100	91	100
Value	Selected_Metric	75	61	65	43
Single_Metric	Value	65	75	44	65
ErrorDialog	ErrorDialog	61	89	27	81

Tab. 8. Comparison at class level between MO_MWB and MO_VM. Couples with $IID_{12} > 60\%$.

$Class_1$	*$Class_2$*	*NL_1*	*NL_2*	*IID_S_{12}*	*IID_S_{21}*	*HM*
New_Class_Metric	New_Sys_Metric	824	776	91	96	94
Class_Custom_Metric_Parser	System_Custom_Metric_Parser	200	147	74	98	84
Function	Method	194	228	74	78	75
Attribute	Variable	110	60	76	67	74
Class	Function	189	194	74	76	73

Tab. 9. Class level functional comparison for self-duplication detection among MO_VM classes, sorted by HM (HM >=70).

The analysis at class level allowed to automatically detect all the classes that have been identified also by the reengineering team, but spending a lower effort for the analysis phase (the time to perform the same analysis has been reduced by 70%). Most of the changes performed by the reengineering team have been also identified by the tool TREND with the metric values. A large part of the classes with a duplication percentage detected by the tool greater than 50% have been strongly modified and reduced by the working team. It is evident that to perform only the analysis at file level cannot result in a reduction of the reengineering time since a deep analysis at class level has to be performed to identify single pieces of code that have to be modified.

$Class_1$	*$Class_2$*	*NL_1*	*NL_2*	*IID_S_{12}*	*IID_S_{21}*	*HM*
UkdmDialog	VMDialog	139	114	62	81	71
CustomMetric	MetricMember	105	198	49	58	59
Single_Metric	Value	34	20	44	65	51

Tab. 10. Class level functional comparison for self-duplication detection among MO_VM classes, sorted by HM (HM >=50).

Once the analysis at class level is performed, an automatic analysis at method level among the identified classes can be also operated in order to identify with a great accuracy the parts of classes that need a massive phase of reengineering without spending time for manual code inspection.

How to interpret metrics and tables

In this section, some comments on how the values in the tables can be read are reported. Note that for brevity only a selection of the complete tables have been reported and therefore not all the duplications detected by the tool are highlighted by the reported tables. The first analysis that has to be typically performed is to verify if structurally similar classes are also similar from the point of view of method implementation. For example, by the comparison of Tab. 7 with Tab. 9, it has been identified that New_Class_Metric and New_Sys_Metric have about 100% of equal attributes, while the functional part is equal at 94%. This is a strong evidence of the duplication of these classes. One of these classes should be eliminated and the other moved in the shared library. Another similar example is that of Function and Method classes or that of Class and Function classes.

The analysis of Tab. 8 can bring to deduce other interesting facts, such as the almost complete functional duplication of classes File, Line, InfoOpen and ErrorDialog. These classes are not present in Tab. 7 because of their low number of attributes. They are in effect duplications of the same class and could be implemented as a unique template.

Several other duplications have been identified by the analysis of the full tables, producing data to be used for the operative phase of the reengineering process. A summary of this data is reported in Tab. 11.

OPERATION TO PERFORM	*NUMBER OF CLASSES*
Deletion	20
Moving in the library	20

Tab.11. Details of the actions to be performed on classes produced by the analysis of results of the tool TREND.

In Tab. 12, the structure of the system after the reengineering analysis performed by means of the tool is compared with the values already reported in Tab. 2. It can be noted that the number of files is increased because it has been hypothesized to split the code in order to have one class for each *hxx* and *cxx* file, and therefore the double of the class number. The number of classes and of their methods and attributes is decreased because the possible duplications have been detected, and therefore eliminated.

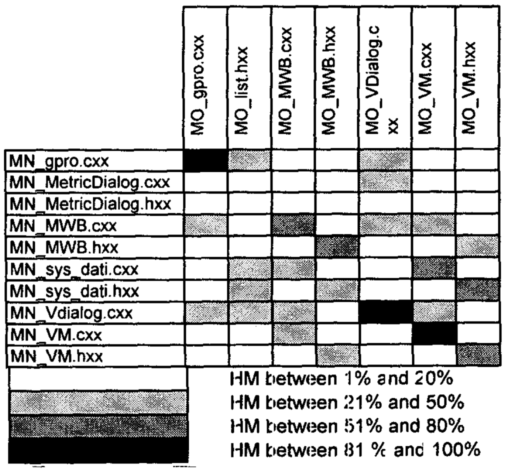
System	*Nfile*	*NCL*	*TNML*	*TNAL*
MWB/VM old	7	76	589	674
MWB/VM new	10	62	530	640
MWB/VM TREND	112	56	~500	~620

Tab. 12. Manual reengineering process against TREND suggestions..

6 ANALYSIS OF REENGINEERING PROCESS

Another useful application of the tool TREND is the analysis of the reengineering process. In particular, with the tool TREND it is possible to analyze the movements of the code during a reengineering by means of an *a-posteriori* analysis. For this kind of analysis a graphical view of the HM values of the table has been adopted. It is very useful to identify which part of the files before the reengineering have moved and where. In Fig. 5, an example of this analysis at file level is reported; the reported Code Movements Table from the old to the new version of the system is evaluated on the basis of HM evaluated on IID_{12} and IID_S_{12} metrics.

In general, the full graph can be interpreted in the following way: taking a row (representing a file in the new system), we can deduce from which file of the old system the source code comes. If the row contains a great part of white boxes and no dark gray or black boxes, the file can be considered completely new, otherwise a certain part of other files has moved the code in it. On the other hand, by analyzing the columns we can deduct that a file of the old version has been eliminated if the column contains only white boxes.



	MO_gpro.cxx	MO_list.hxx	MO_MWB.cxx	MO_MWB.hxx	MO_VDialog.cxx	MO_VM.cxx	MO_VM.hxx
MN_gpro.cxx	■	░	□	□	░	□	□
MN_MetricDialog.cxx	□	□	□	□	░	□	□
MN_MetricDialog.hxx	□	□	□	□	□	□	□
MN_MWB.cxx	░	□	▒	□	░	░	□
MN_MWB.hxx	□	□	□	▒	□	□	░
MN_sys_dati.cxx	□	░	░	□	□	▒	□
MN_sys_dati.hxx	□	░	□	░	□	□	▒
MN_Vdialog.cxx	░	░	░	□	■	░	□
MN_VM.cxx	□	□	░	□	□	■	□
MN_VM.hxx	□	□	□	░	□	□	▒

□	HM between 1% and 20%
░	HM between 21% and 50%
▒	HM between 51% and 80%
■	HM between 81 % and 100%

Fig. 5. Code Movements Table from the old to the new version of the system. evaluated on the basis of HM between the IID_{12} and IID_S_{12} metrics.

For example, file MN_sys_dati.cxx derives its code from MO_MWB.cxx and MO_VM.cxx, while MN_Metric_Dialog.cxx/hxx are quite completely new.

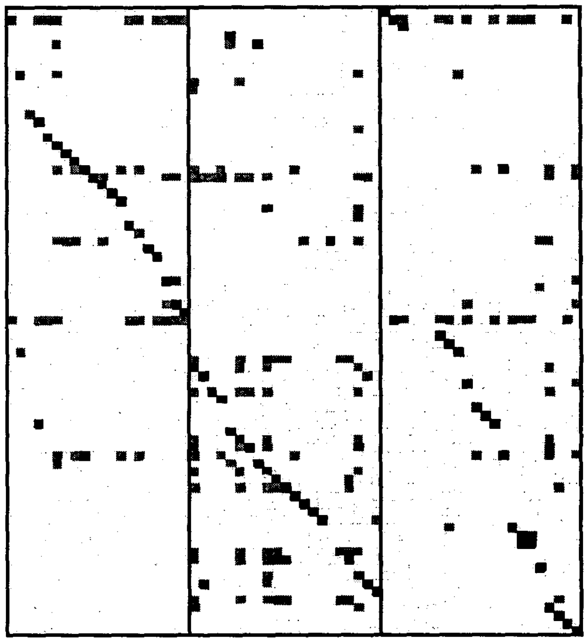


Fig. 6. Comparison at class level between the two versions of the system.

At class level, a similar analysis can be also performed, and the result is depicted in Fig.6, where a more precise estimation of classes evolution between the two versions is reported. The interpretation is similar to that at file level. For this last table the rows and columns represent classes. Classes corresponding to light-colored columns have been eliminated, while classes belonging to light-colored rows have been created as new.

Concluding the analysis, it has been shown how the tool is automatically capable of identifying duplications and aids the reengineering process. The analysis of results provided by the tool has highlighted that not all the possible duplications have been eliminated by the reengineering team and a low level of duplication among classes still exists.

7 CONCLUSIONS

This paper has presented a case study in the field of duplication analysis particularly focussed on the process of reengineering of systems originated in the same application domain and sharing the same class framework or library. A manual approach to the reengineering has been compared against a tool-aided approach, highlighting the time saving (8 man/days against 1 man/month) and the amount of duplications undiscovered by the manual analysis. It has been stated that a tool for duplication detection can drastically reduce the time to identify the duplication in software systems at file, class and method level only if suitable metrics and duplication analysis support are available. On the other hand, the adoption of a tool only at file level does not guarantee a better identification with respect to what can be performed manually by an expert system engineer. The inspection at class and method levels reduces the time to perform the analysis and allows identifying several activities to be performed during the code manipulation phase. In particular the analysis at method level permits identifying where the duplications are present at code level with high precision. The functional duplication may fail several times in the identification of very similar classes in which the code has been manipulated, leaving unchanged the class structure. These classes have to be carefully inspected since the structural similarity may remain undetected by a functional analysis. All these inspections have been supported by means of specific metrics for duplication analysis at structural and functional level. The metric approach guarantees a strong time reduction for the analysis of the huge amount of data that is generated by an automatic tool also for small/medium systems. In conclusion, the adoption of a tool that allows performing structural and functional duplication analysis with specific metrics can improve the reengineering process of the class library or framework by reducing the analysis time and by increasing the quality of the analysis itself.

ACNOLEDGMENT

This work was partially supported by MURST Ex60% govern Ministry of University and Scientific and Technologic Research, and by CESVIT, High-Tech Agency, Center for Software Quality, Florence. A warm thank to the reviewers that have helped us to polish and technically improve the article.

REFERENCES

1. I. D. Baxter, A. Yahin, L. Moura, M. SantaAnna, L. Bier. Clone Detection Using Abstract Sybtax Trees. *In Proc. of ICSM 98.* IEEE 1998.
2. J. Mayrand, C. Leblanc, E. M. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. *In Int. Conf. on Software System Using Metrics.* IEEE Nov. 1996.
3. S. Ducasse, M. Rieger, S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. *In proc. of the Int. Conf. on Software Maintenance*, Sept. 1999, IEEE.
4. K. Kontogiannis, R. DeMori, E. Merlo, M. Galler and M. Bernstein. Pattern Matching for clone and Concept Detection. *Journal of Automated Software Engineering 3*, 1996, Kluwer Academic Publishers.
5. Brenda Baker. On Finding Duplication and Near-Duplication in Large Software Systems. *Working Conf. on Reverse Engineering*, IEEE 1995.
6. Brenda Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, vol 24, 1992.
7. J. Howard Johnson. Substring Matching for Clone Detection and Change Tracking. *In proc. of the int.l Conf. on software Maintence*, 1994.
8. J. Howard Johnson. Identifying Redundancy in Source Code using fingerprints. *In Proc. of CASCON 93*, 1993.
9. Udi Manber. Finding Similar Files in a Large File System. *In PROC. 1994 Winter Usenix Tecnical Conf.*, 1994.
10. Kostas Kontogiannis. Evalutation Experiments on the Detection of Programming Patterns Using Software Metrics. *In Proc. Fourth Working Conf. on Reverse Engineering*, Ira Baxtern, A. Quilici and C. Verhoef, Eds. 1997, IEEE Computer Society.
11. B. Lauge, D. Proulx, E. Merlo, J. Mayrand, J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, *Int.l Conf. on Software Maintenance*, 1997. IEEE.
12. P. Barson, N. Davey, S. Filed, R. Frank, D.S.W. Tansley. Dynamic Competitive Learning Applied to the Clone Detction Problem. *In Proc. of Int. Workshop on Applications of Neural Networks to Telecommunications 2*, Lawrence Erlbaum, Mahwah, NJ 1995.
13. F. Fioravanti, P. Nesi, F. Stortoni. Metrics for Controlling Effort During Adaptive Maintenance of Object oriented System. *In proc of the Intl Conf. on Software Maintenance*, Sept. 1999, IEEE.
14. M. Balazinska, E. Merlo, M.Dagenais, B.Lagüe, K, Kontogiannis. Measuring Clone Based reengineering opportunities. *In Int. Symp. on Software metrics*, November 1999, IEEE.